ENCAPSULAMIENTO Y POLIMORFISMO

ENCAPSULAMIENTO: para proteger a los atributos de un objeto de modificaciones no deseadas se introduce el concepto de **encapsulación**, *ocultamiento* o *abstracción de datos*.

Facilita el mantenimiento y depuración de los programas, ya que los usuarios de una clase sólo necesitan conocer el prototipo de las funciones que puede realizar, y no deben preocuparse sobre cómo están implementadas (el código específico de cada una). Así, si se varía la implementación de una clase pero su interfaz sigue siendo la misma, los usuarios no necesitarán cambiar las líneas de código en las cuales se haya utilizado.

POLIMORFISMO: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre (por ejemplo una función *sum*() que calcule la suma de matrices o de números complejos). Cuál de los métodos será invocado se resuelve en *tiempo de ejecución* cuando un objeto particular lo necesite. Esta característica se denomina "asignación tardía" o "asignación dinámica".

Algunos lenguajes (C++) proporcionan también medios más estáticos (en *tiempo de compilación*) de polimorfismo, tales como las **plantillas** y la **sobrecarga de operadores**.

Encapsulamiento – Funciones *getters* y *setters*

- Las variables de una clase se declaran con un nivel de acceso **private**, y se implementan métodos de acceso **public** que permiten fijar (set) o consultar (get) el valor de las mismas.
- Las **funciones** y **operadores** miembro acceden **directamente** a las variables del objeto implícito y, por medio del nombre y del operador punto, a las del objeto pasado explícitamente (parámetro de la función).
- Los métodos *getters* retornan el valor del atributo sin necesidad de pasar ningún parámetros.
- Los métodos *setters* reciben un parámetro (valor del atributo) y no retornan ningún valor.
- Nombre: anteponer la palabra *get* o *set* a un identificador descriptivo (que en general utiliza una letra mayúscula al inicio de cada palabra), por ejemplo: setNumeroIteraciones(*num*).
- Facilita el entendimiento y mantenimiento del código. Las modificaciones se introducen en la implementación de los métodos (un sólo lugar donde buscar errores!). El usuario de la clase no tiene que modificar sus archivos fuente que compilan y funcionan adecuadamente.
- Algunos IDE (CodeBlocks, Eclipse), incluyen herramientas que incorporan automáticamente la declaración y definición de estos métodos.

Encapsulamiento – Funciones *getters* y *setters*

Ejemplo: En un proyecto logístico de grandes dimensiones (muchos archivo fuente) se tiene una clase Truck donde una de sus características es la capacidad del tanque de combustible:

```
class Truck
{
    public:  double capacity;     ... ;                    // otras variables y métodos públicos  y/o privados
};
```

Si el usuario es de origen norteamericano, habrá realizado todo su programa (probado y en su versión final), asignando un valor de capacidad en galones utilizando la forma directa Truck::capacity = cant_galones.

El programa se vende a usuarios que expresan la capacidad en litros. Una solución sería escribir dos versiones: una para cada sistema. Habría que buscar todas las apariciones de Truck::capacity y decidir si se modifican o no; recompilar todos los archivos fuentes y corregir todos los bugs que presentarán. Otra solución consiste en brindar la posibilidad de configurar el proyecto para que el usuario determine si desea trabajar en litros o galones. Si se hubiera realizado la clase Truck obedeciendo al encapsulamiento:

```
class Truck
{
    public: double getCapacity() ;    … ;
    private: double capacity;         … ;
};
```

```
double getCapacity()
{
    if( Configuration::Measure == Gallons)
        return capacity;
    else
        return (capacity * 3.78);
}
```

Es suficiente con agregar el archivo de cabecera configuration.h en truck.h, permaneciendo **inalterable la interfase** de la clase (es transparente para el usuario). De esta forma se evitan bugs potenciales.

Polimorfismo – Sobrecarga de funciones

- **Sobrecarga (= *homonimia*) de funciones (overloading)**: permite declarar funciones con el *mismo nombre*, cambia la definición. En tiempo de ejecución se llama a una u otra función dependiendo del número y/o tipo de los argumentos utilizados en la llamada. El valor de retorno no influye en la determinación de la función que es llamada.

- **Valores por defecto de los parámetros**: no es necesaria una correspondencia biunívoca entre la lista de *argumentos actuales* (llamada) y la lista de *argumentos formales* (declaración y definición) de una función. Se pueden definir ***valores por defecto*** para todos o algunos de los argumentos formales; que deben ubicarse *al final de la lista de argumentos*. Al invocar la función los argumentos no especificados adoptan sus valores por defecto. Si se omite un argumento deben omitirse también todos los que se encuentren a continuación.

- **Funciones *friend***: es una función que *no pertenece a la clase*, pero que *tiene permiso* para acceder a sus variables y funciones miembro privadas por medio de los *operadores punto* (.) y *flecha* (→). La *declaración* de una función ó clase como friend de otra, debe incluirse *en la declaración de la clase que autoriza el acceso* a sus datos privados (se preserva la seguridad). Se puede hacer indistintamente en la zona public o private. Ninguna función puede autodeclararse amiga y acceder a la privacidad de una clase.

Ej. 01: Param def + Ej. 02: Clase per

Polimorfismo – funciones *friend*

- Las funciones miembro de la clase_friend pueden acceder a las variables privadas de la clase_original, pero no en sentido inverso: las funciones miembro de la clase_original no puede acceder a un dato privado de la clase_friend.

- La implementación de la función friend, no hace uso del operador de ámbito (::) porque no pertenece a la clase. La llamada es directa (no se usan los operadores '.' ó '→').

- Una función puede ser declarada friend de muchas clases, pero sólo puede ser *miembro* de una única clase (en general no es miembro de ninguna).

- Cuando dos clases deben ser declaradas mutuamente friend se debe utilizar una declaración anticipada (**forward declaration**) de una de ellas en el archivo de declaración:

```
class <clase2>;     // declaración anticipada para el compilador
class <clase1>   { /* funciones y variables miembro de clase1*/  friend clase2; };
class <clase2>   { /* funciones y variables miembro de clase2 */ friend clase1; };
```

- Las funciones friend no contienen el argumento implícito **this**.

Polimorfismo – Funciones *friend*

Ejemplo: se quiere calcular la distancia entre las coordenadas de dos objetos de la clase 'punto' mediante una función miembro pública definida por:

```
float punto::distancia(punto pto)                    // el 2º operando está implícito (es el objeto)
{
   return sqrtf( pow(pto.x-x, 2) + pow(pto.y-y, 2) );
}
```

al llamarla se debe utilizar el operador punto del primer objeto y pasarle el segundo punto:

```
float dist = pto1.distancia(pto2);                   // es confusa la notación
```

Se puede mejorar la forma en que se invoca la función si se modifica su declaración en la clase anteponiendo friend, y adaptando la lista de parámetros para que reciba dos objetos de la clase punto para hacer el cálculo. La definición sería:

```
float distancia(punto pto1, punto pto2)              // no utiliza el operador :: y queda más claro
{                                                    // que la función opera sobre dos puntos
   return sqrt( pow(pto1.x-pto2.x, 2) + pow(pto1.y-pto2.y, 2) );
}
```

la llamada se realiza ahora:

```
float dist = distancia(pto1, pto2);                  // es más clara la notación, 2 operandos explícitos
```

Polimorfismo – Sobrecarga de operadores

- Los operadores se sobrecargan (**overload**) como las funciones, sustituyendo con el *nombre* de la función por la palabra ***operator***, seguida del *carácter o caracteres del operador* de C++ al que se le quiere cambiar el ó los (pueden ser varios) comportamientos.
- Se puede modificar la *definición* de un operador pero no su *gramática*: el número de operandos sobre los que actúa, la precedencia y la asociatividad.
- Al menos un operando debe ser un objeto de la clase en la que se ha definido el operador.
- Un operador sobrecargado puede ser miembro o friend de la clase para la que se define, pero nunca las dos cosas a la vez. La definición de una forma u otra es en algunos casos cuestión de conveniencia (o preferencia personal), mientras que en otros está impuesta.
- Se suelen declarar miembros los operadores **unarios** (actúan sobre un solo objeto).
- Los operadores **binarios** que modifican el primer operando (argumento implícito) deben ser siempre miembros; por ejemplo la asignación ('='). En la declaración y en la definición sólo hará falta incluir en la lista de argumentos el segundo operando.

Polimorfismo – Sobrecarga de operadores

- Aunque el operador de asignación no necesitaría retornar ningún valor, en general al sobrecargarlo se hace retornar una referencia al primer operando para permitir la expresión objeto3 = objeto2 = objeto1.

- Si es posible que el primer operando no sea un objeto de la clase el operador debe ser friend (complejo2 = entero + complejo1). Si se hiciera complejo2 = complejo1 + entero no se necesita recurrir al modificador friend. Otro ejemplo se presenta al sobrecargar el operador producto (*) para pre y post-multiplicar matrices por un escalar. En el caso de la pre-multiplicación el operador **debe ser** friend.

- Para los operadores ++ y -- se debe considerar si son pre o post incremento (decremento):

  - clase& operator++ () se refiere al pre-incremento por convención.

  - clase& operator++ (int j) se refiere al post-incremento. La variable j tiene valor 0.

- Los únicos operadores que no se pueden sobrecargar son el *punto* (.), el *if ternario* (?:), el ***sizeof***, el *scope resolution* (::) y *puntero a miembro de un objeto* (.*).

Puntero *this*

El puntero this es una variable predefinida para todas las funciones u operadores miembro de una clase. Este puntero contiene la dirección del objeto concreto al que se está aplicando la función o el operador miembro, se puede decir que *this es un alias del objeto. Se puede acceder a los miembros privados del objeto mediante el operador flecha: this→<atributo>.

En el caso de operadores miembro sobrecargados, el puntero this es la forma que se utiliza para referirse al objeto al que se está aplicando el operador como primer operando. Si el operador debe retornar una referencia al objeto, (=, +=, ++, etc.); se utiliza this para obtenerla. El operador de asignación se **debe** sobrecargar cuando el objeto incluye punteros, caso contrario se haría una copia binaria y se copia la **dirección** del puntero y no los datos.

Las funciones friend que no son miembros de ninguna clase no disponen de este puntero.

La principal diferencia entre utilizar this o funciones set/get para acceder al atributo privado es que mediante el uso de las funciones se pueden realizar operaciones de validación de los valores antes de asignarlos a las variables.

Operadores de conversión (casting)

- **static_cast** (default): comprueba en tiempo de compilación que los tipos de origen y destino son compatibles. Se usa para documentar o enfatizar las conversiones implícitas. Sintaxis:

  static_cast<tipo> (<objeto>);

  La desventaja es que no comprueba que la conversión se realice correctamente, problema que se puede presentar en herencia de clases.

```
class Tiempo
{
  public:
     Tiempo(int h=0, int m=0) :
            hora(h), minuto(m) {}
     void Mostrar()
     {cout << hora << ":" << minutos << endl;}
     operator int() {return hora*60+minuto;}
  private:
     int hora;
     int minuto;
};
int main()
{
   Tiempo Ahora(12,24);
   int minutos;
   Ahora.Mostrar();
   minutos = static_cast<int> (Ahora);
   //minutos = Ahora; // forma implícito
  cout << minutos << endl;
  return 0;
}
```

```
void print (double number)
    {cout << number << endl;}
void print (int number)
    {cout << number << endl;}
void print (int number, string s)
    { cout << number << ", " << s << endl; }

int main()
{
   double d = 12.24;
   int i = 7;
   unsigned int u = 3;
   print(d);
   print(i);
   print(i, "segunda forma");
   print(static_cast<double>(u));        // sin el cast el
                //compilador no sabe a qué función llamar
   return 0;
}
```

Ej. 05: Static cast

Operadores de conversión (casting)

- **dynamic_cast**: se utiliza casi exclusivamente para solucionar problemas en el caso de herencia. Este tipo de conversión garantiza que la misma se realiza sí y sólo sí el objeto es del tipo de destino, por lo que es más lenta que la conversión estática. Únicamente es capaz de comprobar la herencia "hacia abajo", no es capaz de realizar conversiones transversales. Si el operador no puede realizar la conversión devolverá un NULL para punteros ó lanzará std::bad_cast para referencias.

- **const_cast**: es un operador de uso muy específico que sirve para eliminar el modificador const de una **referencia** o un **puntero**. Actúa durante la compilación. Sintaxis: const_cast<tipo> (<objeto>);

  Es importante advertir que el operador no cambia el tipo del operando; no hace que una variable constante pueda volverse no-constante y ver alterado su valor. Es útil para pasar un objeto constante a una función que espera una referencia. Aunque su uso no es común también se aplica al modificador volatile.

```
/***** Ejemplo
struct cosa
{   ... ;  };

void print(cosa&)
{   ... ;  }

int main()
{
   const cosa valores = { . . . };
   //print(valores);  // compila con error:
                      // invalid initialization of reference
   print(const_cast<cosa&>(valores));
   return 0;
}

fin del ejemplo *****/
```

Ej. 06: Const cast

Operadores de conversión (casting)

- **reinterpret_cast**: permite obtener un puntero de un tipo distinto tipo al original, en general incompatibles. El programador debe verificar que el *cast* no produzca resultados inesperados. Sintaxis:

  reinterpret_cast<tipo> (<objeto>);

```
/***** Ejemplo:
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   int x = 0x12dc34f2;
   int *pix = &x;
   unsigned char *pcx;

   pcx = reinterpret_cast<unsigned char *> (pix);
   cout << x << " = " << hex
        << static_cast<unsigned int> (pcx[0]) << ", "   << static_cast<unsigned int> (pcx[1]) << ", "
        << static_cast<unsigned int> (pcx[2]) << ", "   << static_cast<unsigned int> (pcx[3]) << endl;

  return 0;
}

Salida: 316421362 = f2, 34, dc, 12

fin del ejemplo *****/
```

Ej. 07: Reinterpret cast

Composición de clases

Una clase puede contener variables miembro que sean *objetos* de otra *clase* definida por el usuario. El *constructor* de la clase que contiene objetos de otras clases debe llamar a los *constructores de los objetos contenidos*, en caso de no querer utilizar el constructor por defecto (que llamará implícitamente a los constructores por defecto de los objetos declarados como atributos). Los *constructores* de las clases contenidas se llaman de la misma forma que los *inicializadores* de las variables miembro ordinarias: después del carácter ':' (que aparece tras los argumentos) separados entre ellos por comas, y antes del bloque del constructor.

```
<clase_continente> :: <clase_continente> (argumentos)    // constructor clase contenedora
        : <clase_contenida1> (argumentos),                // ':' + constructores clases contenidas
          <clase_contenida2> (argumentos),                // cada uno con sus parámetros
          [otros inicializadores]                         // de variables miembro
{
        asignación de otras variables;                    // si las hubiera, sino el cuerpo queda vacío
}
```